ParsePDB.pm

Package:	ParsePDB.pm
Author:	Benjamin Bulheller
Mail Address	webmaster-.at.-bulheller.com
Website	http://comp.chem.nottingham.ac.uk/parsepdb/ http://www.bulheller.com
Research Group:	Prof. Jonathan D. Hirst School of Physical Chemistry University of Nottingham

Funded by:	EPSRC
Date:	November 2005 – November 2008
Acknowledgments:	Special thanks to Dr. Daniel Barthel for many, many discussions and help whenever needed!

Licence

Contents

1 Foreword

Despite the fact that there are several packages around with the ability to parse PDB files and do funny things with them (e.g. BIOPERL), it looked like there was a lack of really easy ones. Driven by the need of breaking a protein into its subgroups (models and chains), ParsePDB has been coded with the intention to create a package that is powerful enough to handle PDBs with a fair amount of functions but is still easy to handle.

Keeping the complexity at a minimum, a protein can be read, parsed and its chains written into single files with just three commands, which are as easy as `new`, `Parse` and `WriteChains`. Given certain parameters, the atoms and residues can be counted, renumbered and filtered (i.e. just certain elements or residues can be extracted). Most of the command names are designed in such a way that they may take a little time to type, but are easy to remember and meaningful when being read.

The PDB parser is an integral part of the web interface DichroCalc, which can be freely used at http://comp.chem.nottingham.ac.uk/dichrocalc.

Benjamin Bulheller

2 Installation

The PDB parser itself is a Perl package, indicated by the extension .pm. To install the package globally on your system, you can use the provided makefile to copy it to your library path. To do this, login as root and follow the standard routine:

```
./perl Makefile.PL
make
make install
```

Since the package uses `Error.pm` to handle exceptions, this package needs to be installed, too. If it is not already installed then `make` will issue a warning message about that. `Error.pm` can be found at `http://search.cpan.org/` by searching for "Error", was written by Graham Barr and is maintained by Shlomi Fish. The version which was used during the development of the PDB parser was 0.15.

After installation, the parser can be used in a script by including the library with the `use` command:

```
use ParsePDB;
```

The function `use` searches Perl's library path for the given package. If you do not want to install the package globally on your system (for example if you do not possess root permissions), then you can also copy the .pm file to a folder in you home directory. If you for instance collect your packages in `~/bin/perllib/`, you can add this directory via

```
use lib "$ENV{HOME}/bin/perllib";
use ParsePDB;
```

This would also work for `Error.pm`.

Another nice trick is to add the folder the executed script is actually living in. The package `FindBin` is usually included in standard perl installations and sets the variable `$Bin` to the folder of the script, which can then be included via `lib`:

```
use FindBin qw/$Bin/;
use lib "$Bin";
```

3 Nomenclature and Naming Conventions

For the access to the single elements of the PDB (models, chains, residues, atoms or even specific atoms of a certain type) there are some naming conventions which are followed throughout the parser. It is important to differentiate between two things:

- external values

  These values are read directly from the PDB. This means that the first item in a list (a residue or an atom) might not necessarily start at 1, be sequential or that the values change after renumbering the file. Chains might not be accessible via their external ChainID, in case no external ChainID is given (which is quite common).

  Overview of the numbers and labels of a PDB entry and their "name" in the parser:

```
              ModelNumber (external)
              Model (internal)
              |
              |
MODEL         1
ATOM      1   N   LYS A   1      -15.872   7.811  19.851  1.00 76.73
ATOM      2   C   LYS A   1      -15.332   7.443  18.561  1.00 99.86
ATOM      3   CA  LYS A   1      -14.650   6.096  18.757  1.00 72.69
|         |   |    |   |   |
|         |   |    |   |   |
|         |   |    |   |   Residue (internal)
|         |   |    |   |   ResidueNumber (external)
|         |   |    |   |
|         |   |    |   Chain (internal)
|         |   |    |   ChainLabel (external)
|         |   |    |
|         |   |    ResidueLabel => 'LYS'
|         |   |
|         |   AtomType => 'CA'
|         |   Element  => 'C'
|         |
|         Atom (internal)
Race      AtomNumber (external)
```

- internal numbers

  Each item (model, chain, residue, atom) can be accessed via its sequential number in the domain (starting at 0). This number will never change for models or chains, although there is some logical ambiguity for atoms and residues. `Residue 0` in the second chain is also `Residue 20`, if

no chain is specified and the first chain contains 20 residues (taking into account that counting starts at 0). If a `Residue` is specified, the atom number is relative to that residue and can thus change for that very atom, if no chain or no residue is given.

Although it was at first thought to be a nice idea to let internal numbers start at 1, it turned out to be much more versatile to start at 0, like everything else in perl does, too. That way, for instance an array with AtomTypes can be extracted and while looping over it, the current index of the array (e.g. `$AtomTypes[5]`) can be used to extract the respective atom, which possesses the internal number `Atom 5`.

Using the internal numbers is *much* more favorable than using the external identifiers. It is easier to program, the numbers *always* start at 0 and are *always* sequential. In addition to that, one does not need to worry about missing ChainIDs and format errors like that. Furthermore, processing is *much* faster (something like 10 times and more if it comes to accessing residues or atoms), since the external identifiers have to be translated and this translation may moreover be prone to bugs (like finding only the first match if an identifier is used twice for some reason). Inserted residues have the same external number and can only be accessed via the internal number.

4 Initialization of a PDB Object

The parser can be invoked by the command `ParsePDB`. To create a new object, the method `new` is required:

```
$PDB = ParsePDB->new (FileName => $File);
$PDB->Parse;
```

`->Parse` reads the whole PDB and splits it up into its subgroups. The routine for this is as follows:

- Look for blocks divided by MODEL tags. Every MODEL tag causes the parser to regard the following block as a new model, a terminating ENDMDL is not desperately needed.

  If no MODEL tag is found, the protein is regarded as model 0. This is also the default value for most methods, if no model number is specified. Therefore, if no MODEL tags are given or just one MODEL is present, the model number may be omitted.

- Inside each MODEL—ENDMDL domain (or the whole file if no MODEL is found): Every block of ATOM lines is regarded as one chain either
  - until a following TER or
  - a change of the chain ID.

If HETATMs are following the ATOMs, there are two possibilities:

1. one chain with ATOMs and HETATMs if
   - the blocks have the same chain IDs
   - both have no chain IDs and are not separated by a TER.
2. two chains (one with ATOMs and one with HETATMs) if
   - the blocks have different chain IDs;
   - both blocks have no chain IDs and are separated by a TER.

Theoretically, there is the third possibility of the same (defined) chain ID *and* a TER in between. However, according to the PDB manual, a TER marks the end of a chain, thus it is given the higher priority in that case. A TER within a chain is more or less a violation of the PDB format and in terms of the speed of parsing it is much faster to rely on the fact that a TER can only be at the very end of a chain.

All chains inside one model are numbered sequentially as chain 0, 1, etc. and can be requested with this number. If the following check of the chain IDs is successful, they may also be addressed using the actual letter.

Each line of the protein section is split into its entries during parsing. With parameters like `AtomIndex` and `RedidueIndex`, given to `->Get`, it is possible to access and use these readily processed entires directly, please see `->Get` for more information. The line is cut into pieces according to the following scheme, taken from the Protein Data Bank Contents Guide, version 2.1:

Columns	**Field Name**
1–6	Race
7–11	AtomNumber
13–16	AtomType
17	AltLoc
18–20	ResidueLabel
22	ChainLabel
23–26	ResidueNumber
27	InsResidue
31–38	x
39–46	y
47–54	z
55–60	Occupancy
61–66	Temp
67–80	Rest

The field names of the above table can be given to the various methods to filter the contents, for example to retrieve only atoms with a certain AtomType.

- The chain IDs are checked whether
  - every chain has an ID;
  - no duplicate IDs are found within one model.

  Missing or duplicate chain IDs cause a warning message that this can be corrected using `->RenumberChains`. In this case, the chains can only be accessed via their internal number and the parser does not use or accept the real IDs at all.

  The chain IDs are processed case sensitive! (Have a look at 1FNT to see that this is really necessary...)

If the same PDB has to be read again (after changing something in the file, to return to the original version after having renumbered something or for other reasons that require the object to be updated) this can be done using `->Reset` (the object will then be re-parsed automatically):

```
$PDB->Reset;
```

4.1 Parameters explicitly for `->new`

The parameters for `->new` are divided into two groups. The first one consists of parameters, which can only be given to `->new` directly, while the others can also be changed after the initialization of the object. The default values of all of the following switches can be altered in ParsePDB.pm at the very beginning of the code under "Default Values".

- `FileName => "file.pdb"`

  The PDB file including path. The extension .pdb can be omitted.

- `NoHETATM => 0 | 1 (default 0)`

  If set to "1", HETATM lines will be filtered out before parsing the file. This can be handy, if you do not process HETATMs anyway and can save several checks, whether a chain contains any ATOMs at all.

- `NoANISIG => 0 | 1 (default 0)`

  If set to "1", SIGATM, SIGUIJ and ANISOU lines will be filtered out before parsing the file. If you do not process these atoms, it saves processing time (each atom needs to be compared against two strings only instead of five) and avoids checks for you.

4.2 Changeable Parameters for ->new

All the following parameters can be given to ->new or alternatively changed later on in the program using one of the ->SetVariable methods. This is mainly useful to avoid a ->new-command that needs three lines to be viewed entirely...

- ChainLabelAsLetter => 0 | 1 (default 0)

  Tells ->WriteChains whether the exported file names should be named with the number of the chain or the actual chain ID letter. Check whether ->ChainLabelsValid returns true before accessing the chains via letters, otherwise you can only use numbers until the IDs have been corrected with ->RenumberChains!

  ```
  $PDB->SetChainLabelAsLetter (0);
  ```

- ChainSuffix => "-c" (default "-c")

  Defines the suffix that is added to the base name by ->WriteChains

  ```
  $PDB->SetChainSuffix ("-c");
  ```

- ModelSuffix => "-m" (default "-m")

  Defines the suffix that is added to the base name by ->WriteModels

  ```
  $PDB->SetModelSuffix ("-m");
  ```

- HeaderRemark => 0 | 1 (default 1)

  If enabled, a remark that the file has been changed by the parser and the header and footer information might not be valid any more is added to the header. This is done be ->Get and ->Write, but not if the header is requested using the method ->GetHeader.

  By default, the remark lines are added either after HEADER, COMPND or TITLE, depending on which line is found last. That is, the comment will be inserted as the first REMARK lines. If another position is needed, e.g. directly after the HEADER line, then this has to be changed at the beginning of ParsePDB.pm under default values.

  ```
  $PDB->SetHeaderRemark (1);
  ```

- AtomLocations => First|All|None|'A'|'B'| etc. (default 'All')
  Tells ->Get and ->Write globally how atoms with alternative atom locations are to be handled. Please see "->Get" for more information. If you do not need the alternative locations at all, you can use RemoveAtomLocations to get rid of them.

  ```
  $PDB->SetAtomLocations ("All");
  ```

- Verbose => 0 | 1 (default 1)

  Turn verbose mode off or on. If enabled, all warnings (i.e. wrong ChainLabel) are printed.

  ```
  $PDB->SetVerbose (1);
  ```

5 Identify Subgroups of the Protein for the Use in Loops

The Identify-Methods return the requested identifiers of a specific domain in the PDB. This domain (model, chain, residue) can be narrowed using the internal or external identifiers. The returned list can then be used in a loop to use it with `->Get`.

- `->IdentifyModels`

  Returns an array with the internal identifiers of the models, i.e. [0, 1, 2] if three models are present.

  ```
  @AllModels = $PDB->IdentifyModels;
  ```

- `->IdentifyModelNumbers`

  Returns an array with the external identifiers of the models, i.e. [1, 2, 3] if three models are present. These numbers may change, if the file is renumbered via `->RenumberModels`.

  ```
  @AllModelNumbers = $PDB->IdentifyModelNumbers;
  ```

- `->IdentifyChains`

  Returns an array with the internal identifiers of the chains in a certain model, i.e. [0, 1, 2] if three chains are present. If no model is given, 0 is taken by default. These numbers represent the chains in the order as they occur in the PDB. Accessing the chains via their sequential numbers is in any case more secure than using the chain IDs and works definitely with EVERY file, no matter how crappy its format turns out to be. I'm sorry, if I keep on repeating myself, it comes with increasing age, sorry about that.

  ```
  @AllChains = $PDB->IdentifyChains (Model => 0);
  ```

- `->IdentifyChainLabels`

  Returns an array with the chain IDs of the chains in a certain model, e.g. ['A', 'B', 'C'] if three chains are present. If no model is given, 0 is taken by default.

  Check whether `->ChainLabelsValid` returns true before accessing the chains via letters, otherwise you can only use numbers until the IDs have been corrected with `->RenumberChains`!

  It is *strongly* recommended to use the numbers given by `->IdentifyChains` to access the chains rather than using letters!

  ```
  @AllChains = $PDB->IdentifyChainLabels (Model => 0);
  ```

- `->IdentifyResidues`

  Returns an array with all internal residue sequence numbers.

  ```
  @AllResidues = $PDB->IdentifyResidues (Model => 0, Chain => 0);
  ```

- `->IdentifyResidueLabels`

Returns an array with all external residue labels (e.g. ALA, TYR, ...).

```
@ResidueLabels = $PDB->IdentifyResidueLabels (Model => 0, Chain => 0);
```

This array represents the sequence of the amino acids in the requested chain. If one-letter-codes are preferred, the parameter `OneLetterCode` may be set to 1:

```
@ResidueLabels = $PDB->IdentifyResidueLabels (Model => 0,
Chain => 0, OneLetterCode => 1);
```

Mind, that `OneLetterCode` only makes sense, when no hetero atoms are in the PDB, that is for example, `NoHETATM` is set to 1. The method will nevertheless return undef for "unknown" residues like metals or water to ensure the comparability of returned arrays (that for the same search parameters like Model 0, Chain 0, a certain index will always belong to the same residue).

- `->IdentifyResidueNumbers`

  Returns an array with all external residue numbers (including the inserted residue tag if present). Beware of multiple numbers due to the restart of the numbering in every model (or even in every chain, depending on how crappy the file is). To be on the safe side, always specify model and chain or use `->RenumberResidues` prior to `->IdentifyResiduesNumbers`. If model is omitted, 0 is taken as default value, if no chain is specified, all chains are processed.

  ```
  @ResidueNumbers = $PDB->IdentifyResidueNumbers (Chain => 0);
  ```

- `->IdentifyAtoms`

  Returns an array with all internal atom numbers of the requested model, chain or even residue.

  ```
  @AllAtoms = $PDB->IdentifyAtoms (Model => 1, Chain => 0);
  ```

- `->IdentifyAtomNumbers`

  Returns an array with all external atom numbers.

  ```
  @AllAtomNumbers = $PDB->IdentifyAtomNumbers (Chain => 0);
  ```

- `->IdentifyAtomTypes`

  Returns an array with all available atom types (e.g. 'CA', 'CB', 'O') that can be used to filter the atoms with `->Get (AtomType => ...)`.

  ```
  @AllAtomTypes = $PDB->IdentifyAtomTypes (Chain => 0);
  ```

- `->IdentifyElements`

  Returns an array with all available atom elements (e.g. 'C', 'N', 'O') that can be used to filter the atoms with `->Get (Element => ...)`. To refine the filter pattern, you need to edit the filter variables at the very beginning of ParsePDB.pm.

  ```
  @AllElements = $PDB->IdentifyElements (Model => 0, Chain => 0);
  ```

6 Count Number of Subgroups of the Protein

- `->CountModels`

  Returns the number of models in the PDB.

  ```
  $ModelNumber = $PDB->CountModels;
  ```

- `->CountChains`

  Returns the number of chains in a model. If no model is given, 0 is taken by default.

  ```
  $ChainNumber = $PDB->CountChains (Model => 2);
  ```

- `->CountAtoms`

  Returns the number of atoms in the specified part of the protein. If no model is given, 0 is taken by default. ATOM and HETATM lines are treated equally, if you do not want to process HETATMs filter them out via `NoHETATM = 1`, see `->new`.

  ```
  $AtomNumber = $PDB->CountAtoms (Model => 0, Chain => 2);
  ```

  If you need HETATMs but want to determine the number of ATOMs or HETATMs in one model or chain, you can use the parameter `Race => "ATOM"`

  ```
  $AtomNumber = $PDB->CountAtoms (Model => 0, Chain => 2, Race => "ATOM");
  ```

- `->CountResidues`

  Returns the number of residues. If no model is given, 0 is taken by default.

  ```
  $ResidueNumber = $PDB->CountResidues (Model => 0, Chain => 1);
  ```

7 Retrieve a Part of the PDB with ->Get

The method ->Get is the universal tool to retrieve content from the parsed PDB. The information gathered via the Identify or the GetIdentifier methods can be fed into ->Get to retrieve the respective PDB lines.

Although ->Get can handle external identifiers, the access is much more efficient (faster!) via the internal ones, since the former have to be translated before the content can be retrieved.

```
@Chain2 = $PDB->Get (Model => 0, Chain => 0);
```

7.1 Parameters for ->Get

The following Parameters can be used to request a specific part of the protein from ->Get, including several possibilities to change it to certain needs.

- `Model => 0 | 1 | 2 | ...  (internal value)`

  `ModelNumber => 1 | 2 | 3 | ...  (external value from the PDB)`

  The number of the model. The available model identifiers can be retrieved using ->IdentifyModels and ->IdentifyModelNumbers.

- `Chain => 0 | 1 | 2 | ...  (internal value)`

  `ChainLabel => 'A' | 'B' | 'C' | ...  (external value)`

  The number of the chain. The available chain identifiers can be retrieved using ->IdentifyChains. Using the latter method is *only* possible, if the chain IDs have been checked successfully and no duplicate or missing IDs have been found. This means that (if you want to access the chains via their ID) you have to add an if condition to check whether you can do so or not.

  `$PDB->ChainLabelsValid`

  - returns true if the chain IDs are OK;
  - returns false if missing or multiple chain IDs have been detected.

  To get the "real" chain IDs, use ->IdentifyChainLabels, to get the ID for a particular chain, use ->GetChainLabel (see further down).

- `Residue => 1 | 2 | 3 | ...  (internal value)`

  `ResidueNumber => 1 | 2 | 3 | ...  (external value)`

  Returns the ATOM lines of a particular residue

- `ResidueLabel => 'ALA'`

  Returns only alanine residues

- `Atom => 1 | 2 | 3 | ...  (internal value)`

  `AtomNumber => 1 | 2 | 3 | ...  (external value)`

  Returns the ATOM line of a particular atom

- `AtomType => 'CA'`

  Returns only CA atoms. To distinguish between α carbons and calcium, enter "Ca" for the latter. To retrieve only carbons, use `Race => "ATOM"` which filters out all HETATMs.

- `Element => 'C' | 'O' | 'N' | 'H'`

  Return only carbons, oxygens, etc. This will also get "CA" or "OXT". To refine the filter pattern, you need to edit the filter variables at the very beginning of ParsePDB.pm.

- `Header => 0 | 1`

  Include the header true/false. If a parsed content is requested via the `AtomIndex` keyword, the first six characters of the line are stored as `Race` (similar to atoms) and the remaining columns are stored in `Rest`.

  By default the header is NOT included.

- `MinHeader => 0 | 1`

  Include just a minimal header true/false. This is false by default. If set true, only lines beginning with HEADER, TITLE or COMPND are returned. This command overrides the value of HeaderRemark, that is, no remark will be added to a minimal header. If the choice of lines needs to be changed, this can be done in ParsePDB.pm at the beginning of the code under default variables.

  If `MinHeader` is given, the `Header` keyword can be omitted.

- `Footer => 0 | 1`

  Include the footer true/false. By default the footer is NOT included.

- `MinFooter => 0 | 1`

  Include just a minimal footer true/false. This is false by default. If set true, only the line beginning with END is returned. If additional other lines are needed, this can be changed in ParsePDB.pm at the beginning of the code under default variables.

  If `MinFooter` is given, the `Footer` keyword can be omitted.

- `ModelStart => value`

  Renumber the Models in the returned content. This does not affect the main hash, that is, the content is renumbered after it was extracted. To renumber globally, see `RenumberModels` instead.

- `ChainStart => letter`

  Renumber the chain IDs of the returned ATOM and HETATM lines starting with the given letter. The letter is processed case-sensitive. If the letter 'Z' is reached during renumbering, the next chain will be 'a', continuing with non-capital letters. After reaching z the number 0-9 are used. Proteins larger than that require to start again with A-Z. This does not affect the main hash, that is, the content is renumbered after it was extracted. To renumber globally, see `RenumberChains` instead.

- `ResidueStart => value`

  Renumber the residue numbers of the returned ATOM and HETATM lines sequentially starting at 'value' (or 1 by default). See also `RenumberResidues`.

- `AtomStart => value`

  Renumber the returned ATOM and HETATM lines sequentially starting at value or 1 if no value is given. See also `RenumberAtoms`.

- `KeepInsertions => 0 | 1`

  If set to 0, the insertion codes of residues are considered during renumbering the residues. Please read "Filtering the data" for more information.

- `SetChainLabel => ' ' | 'A' | 'B' | ...`

  Changes the chain ID of all returned ATOM, SIGATM, ANISOU, SIGUIJ and HETATM lines. By giving a blank the chain label can be removed.

  CAUTION: This function should be used carefully! It does not care about multiple chains and will simply change every ID to the given value! To "renumber" the chains use the `'ChainStart'`-keyword instead.

- `PDB2CHARMM => 1`

  Formats the returned array for the use with CHARMM. Read further down for more information.

- `CHARMM2PDB => 1`

  For the use with CHARMM-created PDBs, replaces the atom types of CHARMM with the standard PDB labels. Read further down for more information.

- `AtomLocations => First | All | None | 'A' | 'B' | etc.`

  If alternative atom locations for an atom are available, return either only the first, all or none of them. The switches are not case-sensitive.

  If 'First' is requested, the filter returns all atoms with location 'A' as well as the ones, which have NO alternative location.

  If you enter 'A', ONLY atoms with altLoc 'A' are retrieved. If you want to get also the ones with no altLoc, you have to request '(A|)'. Please see "Filtering the data" for more information on

the location indicators in the ATOM line.

- `AtomIndex => 1`

  This can be a very handy command, especially if the parser is only used to retrieve the chains one after another. `AtomIndex` tells `->Get` to return not the original ATOM lines from the PDB but the fully parsed entries. That is, each line (atom) in this array is a hash in which the respective ATOM line is broken down into its parts (see also `->new` for the used scheme).

```
@Content = $PDB->Get (AtomNumber => 5);
Content[0] = 'ATOM      5  CB AVAL A   1       1.224  33.077   8.946  1.00 13.10'

@Content = $PDB->Get (AtomNumber => 5, AtomIndex => 1);
$Content[0] = {
   'Race' => 'ATOM',
   'AtomNumber' => '5',
   'AtomType' => 'CB',
   'AltLoc' => 'A',
   'ResidueLabel' => 'VAL',
   'ChainLabel' => 'A',
   'InsResidue' => '',
   'ResidueNumber' => '1',
   'x' => '1.224',
   'y' => '33.077',
   'z' => '8.946',
   'Occupancy' => '1.0',
   'Temp' => '13.10',
   'Rest' => '     1TBE 113',
}
```

- `ResidueIndex => 1`

  This switch is even more powerful than `AtomIndex` and returns the parsed information of the latter divided up into residues while still providing the same information of `AtomIndex` on a per-residue basis. The parsed information for each individual residue is available and for each its atoms can be accessed either via an array (by looping over them) or via a hash using their atom types as keys.

```
@Content = $PDB->Get (ResidueNumber => 12, ResidueIndex => 1);
$Content[0] = {
   'Atoms'     => array reference with the parsed atom lines,
   'AtomTypes' => hash reference with the parsed atom lines,
   'InsResidue' = ' ',
   'Phi' = '-120.23',
   'Psi' = '104.12',
   'ResidueLabel' = 'ALA',
   'ResidueNumber' = '12'
}
```

  Note that the phi and psi angles are not computed automatically. In order to have them in `@ResidueIndex`, `->GetAngles` (without any parameters) has to be executed once. The angles are then calculated for the whole file and saved.

  In the above example a single residue is retrieved (having the external number 12) and therefore only one array element is returned. This can of course be used for a whole chain to conveniently loop over each residue and each atom within it. The atoms are available in the array

Atoms which comes in handy if every single one needs to be accessed one after another. If particular atoms are needed (like only the C_α carbon for example) they can be accessed via the AtomTypes hash. The same can be achieved directly via ->Get by requesting a specific atom type in a residue, however, if this is needed for every residue in a protein the approach via the ResidueIndex is by orders of magnitudes more efficient.

The hash AtomTypes is only generated if the atom types are unambiguous. That is, if one type is found more than once, the key AtomTypes of this residue will be removed since the integrity of the atom type cannot be assured. If used in a script, the absence of this key points to problems with the atom types and the array can be used instead to find out what is wrong in the PDB.

The following example shows the result of the query

```
@Content = $PDB->Get (ResidueNumber => 1, ResidueIndex => 1);
```

for a residue containing two atoms with the atom types CA and O:

```
$Content[0] = {
  'AtomTypes' => {
                   'CA' => {
                             'AltLoc' => 'A',
                             'AtomNumber' => '3',
                             'AtomType' => 'CA',
                             'ChainLabel' => 'A',
                             'InsResidue' => ' ',
                             'Occupancy' => '0.0',
                             'Race' => 'ATOM',
                             'ResidueLabel' => 'ALA',
                             'ResidueNumber' => '1',
                             'Rest' => '           C  ',
                             'Temp' => '27.84',
                             'x' => '-10.309',
                             'y' => '53.910',
                             'z' => '25.295'
                           },
                   'O' => {
                             'AltLoc' => 'A',
                             'AtomNumber' => '4',
                             'AtomType' => 'O',
                             'ChainLabel' => 'A',
                             'InsResidue' => ' ',
                             'Occupancy' => '0.0',
                             'Race' => 'ATOM',
                             'ResidueLabel' => 'ALA',
                             'ResidueNumber' => '1',
                             'Rest' => '           O  ',
                             'Temp' => '27.80',
                             'x' => '-9.414',
                             'y' => '53.366',
                             'z' => '24.654'
                           }
                 },
  'Atoms' => [
               {
                 'AltLoc' => 'A',
                 'AtomNumber' => '3',
                 'AtomType' => 'CA',
                 'ChainLabel' => 'A',
```

```
                    'InsResidue' => ' ',
                    'Occupancy' => '0.0',
                    'Race' => 'ATOM',
                    'ResidueLabel' => 'ALA',
                    'ResidueNumber' => '1',
                    'Rest' => '          C  ',
                    'Temp' => '27.84',
                    'x' => '-10.309',
                    'y' => '53.910',
                    'z' => '25.295'
                  },
                  {
                    'AltLoc' => 'A',
                    'AtomNumber' => '4',
                    'AtomType' => 'O',
                    'ChainLabel' => 'A',
                    'InsResidue' => ' ',
                    'Occupancy' => '0.0',
                    'Race' => 'ATOM',
                    'ResidueLabel' => 'ALA',
                    'ResidueNumber' => '1',
                    'Rest' => '          O  ',
                    'Temp' => '27.80',
                    'x' => '-9.414',
                    'y' => '53.366',
                    'z' => '24.654'
                  }
                ],
  'InsResidue' => ' ',
  'ResidueLabel' => 'ALA',
  'ResidueNumber' => '1'
};
```

With `@ResidueIndex` it is easy to loop over the whole chain residue per residue and even over each atom in the residue:

```
@ResidueIndex = $PDB->Get (Chain => 0, ResidueIndex => 1);

foreach $Residue (@ResidueIndex) {
   print "$Residue->{ResidueLabel}\n";
   print "$Residue->{ResidueNumber}\n";
   print "$Residue->{Phi}   $Residue->{Psi}\n";

   foreach $Atom ( @{$Residue->{Atoms}} ) {
      print $Atom->{AtomNumber}, "\n";
      print $Atom->{ChainLabel}, "\n";
   }
}
```

The TER is not counted as part of the residue, that is it will not be included in the last residue of a chain.

The filter arguments are accumulative and can be freely combined to filter out all CA atoms in all alanine residues for instance. They are just working on the atom lines, all other lines like MODEL, TER, etc. are also returned.

The `->Get` routine returns the whole PDB with the MODEL/ENDMDL tags and one single Model without them but with the TER terminators of the chains.

If the ”Model” or ”Chain” parameter is omitted, it depends what happens:

- no Model, no Chain:
  returns the whole file (without header and footer)
- Model, but no Chain:
  the whole model with all chains is returned
- Chain, but no Model:
  Model is assumed as ”0”, e.g. if no MODEL tags are there at all

7.2 Internal Versus External Identifiers

As previously mentioned, the use of internal identifiers (see 3) is preferable. First of all it is easier to program as e.g. the `ResidueNumbers` or `AtomNumbers` have to be retrieved at first to loop over them and extract single objects. And second of all for speed reasons, as it is quite elaborate for the parser to interconvert external and internal identifiers.

Atoms and residues always have an external number, however, PDBs with invalid numbering schemes are even found in the Protein Data Bank. The worst case are double atom numbers, in which case only the first atom is returned if it is tried to access them via the `AtomNumber`. For residues, the numbers may also contain letters in case of inserted residues like `34A`. In most of the PDB files, only one model exists and this is usually not explicitly named, thus it possesses no external number.

The biggest problem are chain IDs. Very often, chains are not named at all (empty chain ID) or double IDs exist (like A, B, A, B). In both cases, `ChainLabel` cannot be used to access the chains. Every used model is checked right after parsing for those errors. Each use of `ChainLabel` triggers a check whether the chain IDs of the current model have passed this check and are valid. If they are invalid, a warning is issued, the parser stops processing and returns `undef`.

It is also possible for the user to determine, whether external chain identifiers can be used for a model:

```
if ($PDB->ChainLabelsValid (Model => 0)) { ... }
                                    else { ... }
```

As usual, the method defaults to model 0, if no parameters are given. If the chain labels are invalid, the chains need to be renumbered, before they can be accessed via `ChainLabel`.

8 Write the Whole or Parts of the PDB

To write out specific parts of the PDB, the method ->Write is used. All parameters are identical with ->Get, so it is possible to write out for example only certain atom types or all alanine residues and so on.

```
$PDB->Write (FileName => "file.pdb");
```

Header and Footer are included by default. The parameter FileName is mandatory, if it is omitted, the method throws a NoFile error.

Sometimes it is required to split a protein into its models or chains. Since that is a standard task, there are two methods for it.

```
@AllModels = $PDB->WriteModels; # extract all models in single files
```

```
@AllChains = $PDB->WriteChains; # extract all chains in single files
```

Most parameters of these two methods are identical with ->Get. Header and Footer are included by default. If the parameter "FileName" is omitted, the base name of the PDB is taken.

The suffix defined via ->new (ModelSuffix or ChainSuffix) is added plus the chain or model number, for example:

```
file_c1.pdb
file_c2.pdb
file_c3.pdb
```

An array containing the names of the created files is returned.

->WriteChains can also write the chain IDs as letters instead of numbers. Numbers are the default, if letters are desired, ChainLabelAsLetter has to be set to 1 via ->new() or ->SetChainLabelAsLetter. If ->ChainLabelsValid returns false, the setting is ignored and chains can only be accessed via their sequential numbers given by ->IdentifyChains.

If no Model is given to ->WriteChains, "0" is taken by default. The routine can just process one model at a time. To loop over all models, the information provided by the method ->IdentifyModels can be used. When all models are processed, something like "FileName => "$BaseName-$Model" should be defined as file name.

The following (additional) parameters can be given to ->WriteModels and ->WriteChains.

- ModelSuffix => "-model-"

  To change the default suffix "-m".

- ChainSuffix => "-chain-"

To change the default suffix "-c".

- `PDB2CHARM => 1`

  To write CHARMM-Input-Files. Read further down for more information.

- `CHARMM2PDB => 1`

  To convert CHARMM-generated PDB format to the PDB standard. Read further down for more information.

9 Retrieve Certain Information About the Protein

- `->GetModel`

  Returns the internal number of an external `ModelNumber`

  ```
  $Model = $PDB->GetModel (ModelNumber => 4);
  ```

- `->GetModelNumber`

  Returns the external number of an internal `Model`

  ```
  $ModelNumber = $PDB->GetModelNumber (Model => 1);
  ```

- `->GetChain`

  Returns the internal number of an external ChainLabel

  ```
  $Chain = $PDB->GetChain (Model => 1, ChainLabel => 'A');
  ```

- `->GetChainLabel`

  Returns the "real" chain ID for a particular chain, `undef` if no ChainLabel is set.

  ```
  $ChainLabel = $PDB->GetChainLabel (Model => 1, Chain => 1);
  ```

- `->GetResidue`

  Returns the internal number of an external `ResidueNumber`.

  ```
  $Residue = $PDB->GetResidue (Model => 1, ResidueNumber => 5);
  ```

- `->GetResidueLabel`

  Returns the label of the residue with a given residue number (usually the amino acid). The number can be either the external one indicated by the keyword `ResidueNumber`" or the internal one by giving the `Residue` keyword.

  The available numbers can be retrieved via `->IdentifyResidues` and `->IdentifyResidueNumbers` and the returned values can be fed into `->Get` via `Residue => "value"` or `ResidueNumber => "value"` respectively.

  ```
  $ResidueLabel = $PDB->GetResidueLabel (Residue => 2);
  ```

  ```
  $ResidueLabel = $PDB->GetResidueLabel (ResidueNumber => 2);
  ```

- ->GetAtom

  Returns the internal number of an external AtomNumber.

  ```
  $Atom = $PDB->GetAtom (AtomNumber => 17);
  ```

- ->GetAtomNumber

  Returns the external number of an internal atom number.

  ```
  $AtomNumber = $PDB->GetAtomNumber (Atom => 0);
  ```

- ->GetAtomType

  Returns the type of the atom with a given atom number. The available numbers can be retrieved via ->IdentifyAtoms and the returned value can be fed into ->Get via AtomType => "type".

  ```
  $AtomType = $PDB->GetAtomType (Atom => 2); # internal

  $AtomType = $PDB->GetAtomType (AtomNumber => 3); # external
  ```

- ->GetElement

  Returns the element of the atom with a given atom number. The available numbers can be retrieved via ->IdentifyAtoms and the returned value can be given to ->Get via Element => "element".

  ```
  $Element = $PDB->GetElement (Atom => 5);
  ```

  If you plan to use the ResidueIndex or AtomIndex later on and need the Element, you can call the routine without parameters, which will save the element information to the main hash, that is it will then be available in the returned ResidueIndex and AtomIndex, respectively:

  ```
  $PDB->GetElement;
  ```

- ->GetCoordinates

  Returns a 2-dimensional array with the coordinates of the requested atoms (all parameters like Model, Chain and the filter commands are given to ->Get and all lines not beginning with ATOM (e.g. TER) are ignored).

  CAUTION: All coordinates are returned, regardless of how many chains are retrieved from ->Get. So be careful to specify a particular one!

  ```
  @Coordinates = $PDB-$>$GetCoordinates (ChainLabel => 'A');
  print "First Atom:  x $Coordinates[0]->{x},
                      y $Coordinates[0]->{y},
                      z $Coordinates[0]->{z}\n";
  ```

- ->GetAngles

Returns a hash with the Φ and Ψ angles of a chain or a residue. Remember that for the first residue of a chain, no Φ angle is defined whereas the last one does not have a Ψ angle (the angles are then given as 360°).

The angles are returned in an array (if only one residue was processed, only the first element is filled). The routing calculated the bond distance between two residues. If it is too big (for example between two chains or if a residue has been removed for same reason), the respective angle is given as 360°.

```
%Angles = $PDB->GetAngles (Residue => 2);

%Angles = $PDB->GetAngles (ResidueNumber => 2);

print "Phi angle:  $Angles[0]{Phi}\n";

print "Psi angle:  $Angles[0]{Psi}\n\n";
```

If a certain model or chain is processed with the method, the calculated data is only returned, but not saved for later use. Sometimes it is more handy (and faster) to simply calculate all angles in one go and retrieve them later on with other information.

This can be achieved by calling the method without any parameters:

```
->GetAngles; # to calculate and save all dihedral angles
```

The computed angles are then included in the `ResidueIndex` which can be retrieved from `->Get`.

- `->GetSection`

  To fetch certain information from header or footer. All lines starting with the given pattern are returned.

  ```
  @Section = $PDB->GetSection ("CONECT");
  ```

- `->GetResolution`

  Returns the resolution in Angstroms that was used for building the model. If no REMARK 2 field or no resolution is given, `undef` is returned.

  ```
  $Resolution = $PDB->GetResolution;
  ```

- `->GetHeader`

  Returns the header (everything until the first MODEL or ATOM). The parameter `MinHeader => 1` can be given to get only a minimal header.

  ```
  @Header = $PDB->GetHeader;
  ```

- `->GetMinHeader`

  This is a shortcut to retrieve a minimal header.

  ```
  @MinHeader = $PDB->GetMinHeader;
  ```

- `->GetFooter`

  Returns the footer (everything after the last ATOM, HETATM, TER or ENDMDL). The parameter `MinFooter => 1` can be given to get only a minimal footer.

  ```
  @Footer = $PDB->GetFooter;
  ```

- `->GetMinFooter`

  This is a shortcut to retrieve a minimal footer.

  ```
  @MinFooter = $PDB->GetMinFooter;
  ```

10 Renumbering Entries in the PDB

To renumber the protein globally, the following methods can be called directly:

```
$PDB->RenumberModels (ModelStart => 3);
```

```
$PDB->RenumberChains (ChainStart => 'G');
```

```
$PDB->RenumberResidues (ResidueStart => 5);
```

```
$PDB->RenumberAtoms (AtomStart => 5);
```

When renumbering chains, the letter is processed case-sensitive. If the letter 'Z' is reached during renumbering, 'a'-'z' will be used and '0'-'9' after non-captials have been used up. Proteins larger than that require to start again with A-Z, which leads to duplicate chain labels and some routines (e.g. the retrieval of chains using the chain label) will not work then. However, this is a shortcoming of the PDB standard with the chain label being restricted to one character. The parser will not return a chain based on its chain label, if multiple possibilities are found.

If the `Start`-parameters are omitted, 1 and 'A' (for chains) are taken by default.

Be aware that after renumbering, some information in the header and footer does not fit to the atom numbers any more (e.g. SSBOND, HELIX, CONECT)! The numeration of chains restarts in each model.

If the main content should not be altered or for instance every retrieved chain must start at 1, the `Start` parameters can be given to `->Get`, which then renumbers only the filtered content but not the main hash itself. That way, one can extract all alpha carbon atoms and still have a sequential numbering:

```
@Content = $PDB->Get (AtomType => 'CA', AtomStart => 5)
```

Only the returned array is renumbered, leaving the internal data untouched.

10.1 Renumbering Inserted Residues

When inserted residues are contained in the PDB, they usually have the same residue number as the residue before with an added inserted residue tag, e.g. residue 21 and 21A. If the insertion codes should be considered, that is if you want to preserve this numbering scheme explicitely, you can set the parameter `KeepInsertions` to 1.

```
$PDB->RenumberResidues (KeepInsertions => 1);
```

Since '1' is the default for `KeepInsertions`, it is more likely needed when you need to turn it off if you want to remove the inserted residue tags and have each residue numbered with a different number. Be aware, that inserted residues might be superpositions instead of insertions. `->RemoveInsertedResidues` checks the distances of the atoms of each residue with an InsResidue tag and removes only the ones which are indeed superpositions, whereas real insertions are kept. That means, after you have executed `RemoveInsertedResidues`, you can safely renumber the residues discarding their InsResidue tag with

```
$PDB->RenumberResidues (KeepInsertions => 0);
```

10.2 Ignoring the TER

According to the PDB manual, a `TER` line has its own atom number, i.e. it counts as an atom. If you for some reason need all atoms to be numbered sequentially without counting the TER, you can set `IgnoreTER` true. The TER line will then have the same atom number as the atom before:

```
$PDB->RenumberAtoms (IgnoreTER => 1);
```

Please see also further down chapter 12, "Filtering the data".

11 Generating CHARMM input files

Working with CHARMM usually is a pain somewhere where you don't want it. At least the parser is able to solve some of the problems. The main pain is that CHARMM can only process one chain at a time. Although it is possible to give the `"PDB2CHARMM => 1"` parameter to `->Get` and every related method, you will use `->WriteChains` most of the time.

If you use another method, please remember to work with only one chain to produce a valid input file.

```
@CHARMMFiles = $PDB->WriteChains (PDB2CHARMM => 1);
```

To convert the CHARMM PDB output files to standard format, use the realted method `CHARMM2PDB` equivalently with `->Write` or `->Get`.

11.1 What it does

- All chains are written into seperate files
- All residues are renumbered sequentially starting at 1 in each chain
- "HIS" is replaced with "HSD"
- the last O is renamed OT1, OXT is renamed OT2
- All atom types are replaced according to this list:
  `http://www.bmrb.wisc.edu/ref_info/atom_nom.tbl`

When more than one chain is detected, a warning will be issued that CHARMM is unable to process more than one chain at a time.

11.2 What it does not do

- Terminal acetyl groups

  Sometimes proteins have a terminal acetyl group on the amino end which needs to be patched to work with CHARMM. It might also be that it needs to be counted to the first residue (usually it is numbered as residue 0). In that case, `ResidueStart => 0` can be given.

- Prosthetic Groups

  Sometimes PDB files contain coordinates for non-peptide groups (crystallographic waters, haem groups, metal ions, etc). CHARMM can deal with these, but it can be very difficult to do. If you need to include them, split them up into their own PDB files. That means that you might even have to split up a block of HETATMs into several files (e.g. waters, haem groups and other stuff).

- Disulphide bridges

  PDB files hold information on disulphide bridges with the SSBONDS keyword. CHARMM ignores this. You must specifically add these using the PATCH command, for example

  ```
  PATCH DISU prot 2 prot 11
  ```

  creates a disulphide bond between residues 2 and 11 of the protein with the segment id "prot".

  CAUTION: The residue number might have changed since they are auto- matically renumbered so that they start counting at 1!

- Protonation state

  You should consider carefully the protonation state of your titratable residues, e.g. a histidine could be protonated, in which case it should be changed from HSD to HSE. Assuming residue 29 in a segment named "prot" was a histidine that should be protonated, the following patch command would accomplish this:

  ```
  PATCH hs2 prot 29

  rename resn HSE sele resi 29 end
  ```

  There's no easy way to decide if a residue should be modified. One way is to check if there are any H-bond donors or acceptors close to specific atoms (e.g. ring nitrogens in histidine). There may also be comments in the PDB file.

12 Filtering the Data

12.1 Keywords for filtering actions

The returned data of `->Get` and `->Write` can be filtered using the keywords `Model`, `ModelNumber`, `Chain`, `ChainLabel`, `Residue`, `ResidueNumber`, `ResidueLabel`, `Atom`, `AtomNumber`, `AtomType`, `Element` and `Race`. Please see `->Get` for more information.

```
ATOM    554  CA  ILE B   4      55.013  57.563  15.473  6.00  5.58
|          |  |    |  |   |
|          |  |    |  |   |
|          |  |    |  |   Residue (internal)
|          |  |    |  |   ResidueNumber (external)
|          |  |    |  |
|          |  |    |  Chain (internal)
|          |  |    |  ChainLabel (external)
|          |  |    |
|          |  |    ResidueLabel => 'ILE'
|          |  |
|          |  AtomType => 'CA'
|          |  Element  => 'C'
|          |
|          Atom (internal)
|          AtomNumber (external)
Race
```

12.2 Inserted Residues

If evolution has found it clever for some reason to insert some residues in a protein, we're left with a mutant that has the same amino acid sequence as its 'parent' except for a few residues somewhere in the middle. To maintain comparability, these additional residues often have the same residue number with an additional letter to distinguish between them.

When renumbering the residues with the parameter `'KeepInsertions => 0'`, the insertion codes will be removed and all residues numbered with a different number. `'KeepInsertions => 1'` (default) will keep the code and the same residue number.

```
ATOM    157  N   ARG    36       5.498  21.150  28.984  1.00 14.69
ATOM    158  CA  ARG    36       6.066  21.087  27.635  1.00 14.20
ATOM    159  C   ARG    36       5.153  20.325  26.712  1.00 15.33
ATOM    160  O   ARG    36       4.720  19.206  27.052  1.00 16.06
ATOM    161  CB  ARG    36       7.437  20.435  27.542  1.00 16.13
ATOM    162  CG  ARG    36       8.490  21.213  26.764  1.00 21.10
ATOM    163  CD  ARG    36       9.474  20.233  26.120  1.00 24.12
ATOM    164  NE  ARG    36       9.840  19.039  26.969  1.00 23.22
ATOM    165  CZ  ARG    36       9.566  17.748  26.564  1.00 23.96
ATOM    166  NH1 ARG    36       8.793  17.540  25.474  1.00 24.64
ATOM    167  NH2 ARG    36       9.990  16.634  27.226  1.00 23.31
ATOM    168  N   SER    36A      4.771  21.064  25.675  1.00 32.40
ATOM    169  CA  SER    36A      3.791  20.697  24.655  1.00 34.00
ATOM    170  C   SER    36A      4.435  20.746  23.257  1.00 36.06
ATOM    171  O   SER    36A      4.539  21.844  22.669  1.00 34.20
ATOM    172  CB  SER    36A      2.734  21.806  24.772  1.00 32.84
ATOM    173  OG  SER    36A      1.709  21.554  23.828  1.00 40.73
ATOM    174  N   GLY    36B      5.196  19.668  22.952  1.00 29.69
ATOM    175  CA  GLY    36B      6.090  19.571  21.796  1.00 28.11
ATOM    176  C   GLY    36B      7.416  20.232  22.102  1.00 29.54
ATOM    177  O   GLY    36B      8.103  19.879  23.089  1.00 29.74
ATOM    178  N   SER    36C      7.580  21.333  21.367  1.00 34.68
ATOM    179  CA  SER    36C      8.647  22.291  21.565  1.00 34.99
ATOM    180  C   SER    36C      8.085  23.567  22.177  1.00 33.49
ATOM    181  O   SER    36C      8.667  24.643  22.026  1.00 35.29
ATOM    182  CB  SER    36C      9.228  22.641  20.183  1.00 36.90
ATOM    183  OG  SER    36C     10.437  23.401  20.316  1.00 40.81
                             |
                             Inserted residues A, B, C (column 27)
```

A big problem which has to be taken into account are PDB files, which use the inserted residue tag to mark *alternative* residues, that is, residues superimposed with others like ALA and LEU at the same position for example. Without checking the atom distances or the header remarks, it is not possible to distinguish between inserted residues and alternative residues.

If it is important, that the chain has no such alternatives, they can be reliably removed using the method `->RemoveInsertedResidues`. It loops over all residues with an inserted residue tag, checks the atom distances with the neighbouring groups and removes only residues if they are superpositions.

```
$PDB->RemoveInsertedResidues;
```

If it is *really* important that there are no superpositions and the PDB file might be crappy enough that the inserted residue tags are not reliable, the parameter `Intensive` can be set true, what will then cause a check off *all* residues in the protein.

```
$PDB->RemoveInsertedResidues (Intensive => 1);
```

If superpositions are found, the residue with an InsResidue tag is removed, it does not matter, whether the sequence is 36, 36A or the other way round. If none of them possesses an InsResidue marker, the second one is removed.

In all cases, a warning is issued, stating the `ResidueNumbers` (the external ones for easy comparison in the PDB file) and which of them was removed.

12.3 Alternative Atom Locations

If some atoms showed a deviation during the structural elucidation of the protein via NMR or X-Ray, the alternative locations are sometimes stated within the same model, instead of in a second one. The alternate location indicator is usually 'A' or 'B', but has also been found as '1' and '2'. The parser detects the used method automatically.

```
ATOM    143  N   SER    18       7.902   9.621  14.878  1.00  8.15
ATOM    144  CA  SER    18       6.436   9.552  14.567  1.00 10.68
ATOM    145  C   SER    18       6.287   9.730  13.049  1.00 10.82
ATOM    146  O   SER    18       7.124  10.246  12.306  1.00 11.90
ATOM    147  CB  SER    18       5.687  10.570  15.337  1.00 14.98
ATOM    148  OG ASER    18       6.225  11.165  16.468  0.50 14.28
ATOM    149  OG BSER    18       6.181  11.830  15.086  0.50  9.85
                |
                Alternative atom location A and B (column 17)
```

The alternative atom locations can be removed using the AtomLocations keyword via `->new` or `->SetAtomLocations`.

```
$PDB->SetAtomLocations ('First');
```

This will enable the filtering of the additional atoms and return only the first one. If these atoms are not needed anyway, they can be removed entirely with `->RemoveAtomLoactions` and the same keyword, which denotes the atoms which are kept:

```
$PDB->RemoveAtomLocations (AtomLocations => 'First');
```

This deletes all atoms with multiple locations and keeps just the first one. See `->new` for more information. If no parameter is given, the default is taken, which is set to "All" and therefore does not do anything.

13 Other methods

13.1 ->GetFASTA

Returns an array with the FASTA format lines. A model number has to be provided, otherwise model 0 is taken as default. To process only a single chain, `Chain` or `ChainLabel` can be specified. A standard header line is added before a new chain begins and a newline character is added if the sequence line with the one letter codes of the amino acids is 80 characters in length.

The method only processes a chain if there are ATOMs in it, that is, HETATM chains are ignored. If a residue label cannot be converted into the one letter code, a warning is issued by AminoAcidConvert.

```
@FASTA = $PDB->GetFASTA (Model => 0);
```

13.2 ->WriteFASTA

Writes a FASTA file of the protein. If a file name is provided, the extension .fasta is added. If the file name is omitted, the base name of the original PDB file is taken. The method returns the array with the FASTA lines, if they are needed for anything else (and if only for checking whether anything was written at all).

```
->WriteFASTA (Model => 0, FileName => "4mbn")
```

13.3 ->AminoAcidConvert

Converts a 1-letter-code into a 3-letter code and vice versa.

```
$Code = $PDB->AminoAcidConvert ($Code);
```

13.4 ->FormatLine

Returns a string in PDB format from a given atom hash. For many problems working with the Atom-Index is by far easier and faster than retrieving the complete PDB line, e.g. for more complex filtering actions than possible via ->Get. However, if a PDB needs to be written, retrieving the formatted PDB line as well is circumstantial and time consuming, especially if it needs to be tweaked somehow. In such a case, the filtered and/or changed atom hash from the atom index can be given to the routine, which returns the formatted line that can be written to a PDB.

```
@AtomIndex = $PDB->Get (Model => 0, Chain => 0, AtomIndex => 1);

foreach $Atom ( @AtomIndex ) {
```

```
   # some fancy if-condition here or changes to the atom hash
   $Line = $PDB->FormatLine (Atom => $Atom);
   print PDB $Line;
}
```

14 Speed issues

There are a few things you should know, if speed is an issue for your work. As long as you work with a single file of a normal size, the way how you use the parser will not make a big difference. As soon as you have to process a load of PDBs, with ten thousands of atoms, you might want to consider some facts of the way the PDB is treated during the parsing.

First of all, remove everything you do not need. If you do not process HETATM, SIGATM, ANISOU or SIGUIJ entries, remove them even before parsing with the respective switches of `->new` or the respective `->SetVariable` methods. This speeds up the parsing, even if none of these atoms are present, since the regular expressions which determine an atom become significantly smaller. And RegExes are pigs when it comes to speed...

The use of external numbers is an absolute no-no in speed critical programs. Each external parameter is converted to the respective internal one prior to processing the request. This is reasonably fast for models and chains, but comes down to a loop over the atom lines for a residue or an atom number, until the correct one is found. This could have been solved faster by a hash key for each external number but this would raise lots of problems with inserted residues, alternative atom locations, crappy PDB files and would extremely slow down many methods like "Renumber" for example and the idea was therefore discarded.

The access of whole chains is the fastest of all (compared to models, residues and atoms). In other words, the parser is quite optimized for that. Thus, extracting each single atom one by one would be about an order of a magnitude slower than to extract the whole chain and looping over the lines in it. If a residue is requested, the parser has to determine the chain, then looks up the atoms which belong to that residue, extracts and filters the lines and returns them. In any case it speeds up processing if the chain is specified, so rather than looping over the whole model, loop over each chain and process the residues or atoms.

Two of the most powerful switches of `->Get` are `AtomIndex` and `ResidueIndex`. They provide the possibility to extract the readily parsed atom lines instead of the original strings and break a chain down into residues in just one parser query. Therfore, using these switches you can spare cutting the line yourself with `substr` or requesting lots of information like `AtomType`, `AtomNumber` and so on with the respective `->Get` method.

As a ready-made code snippet, the fastest way to access each atom line in a file is the following:

```
# if the dihedral angles are needed
$PDB->GetAngles;

@Models = $PDB->IdentifyModels;
foreach $Model (@Models) {
   @Chains = $PDB->IdentifyChains (Model => $Model);

   foreach $Chain (@Chains) {
      # if the residues are not needed to be processed as a whole
        @AtomIndex = $PDB->Get (Model => $Model, Chain => $Chain, AtomIndex => 1);

        foreach $Atom (@AtomIndex) {
           # do some stuff with each atom
           print "$Atom->{x}   $Atom->{y}   $Atom->{z}\n"
        }

      # ========================================================================

        # or alternatively, if the residues need to be processed one by one
        @ResidueIndex = $PDB->Get (Model => $Model, Chain => $Chain, ResidueIndex => 1);

        foreach $Residue (@ResidueIndex) {
           # do some stuff with each residue
           print "$Residue->{Phi}   $Residue->{Psi}\n\n";

           foreach $Atom ( @{$Residue->{Atoms}} ) {
              # do some stuff with each atom
              print "$Atom->{x}   $Atom->{y}   $Atom->{z}\n"
           }
        }
   } # of foreach $Chain
} # of foreach $Model
```

15 Error handling

The error handling of ParsePDB consists of two levels:

- warnings which are reported and give hints for possible problems but do not cause the processing to be aborted
- errors which are fatal and cause ParsePDB to abort the processing

If you are not familiar with the try/throw/catch-methodology have a look at the documentation of `Error.pm`. In case you do not make use of this, the program dies as any other program does, issuing the respective error message. However, if you catch the error in the main program, it will "survive" and can handle the error appropriately (or simply close open file handles before dieing off as well).

15.1 Which error can happen where?

This listing can be helpful to decide for which error to check after several actions in your script. As you can see, this is merely necessary after `->new` (whose only error is due to user incompetence) and `->Parse`. After that, the PDB and its parser should work smoothly together...

- `->new`

  `NoFile`

  If no file name was given or the file has not been found

  `FileNotFound`

  If the file given via `FileName` was not found.

- `->Parse` and `->Reset`

  `IOError`

  If the file could not be opened, e.g. due to a permission problem.

  `CorruptFile`

  If the file is empty or no ATOM lines have been found in it

- `->Get` and `->Write` (and all methods using them)

  `UnknownElement`

  If a specific element was requested via `'Element => '` that has not yet been specified in ParsePDB.pm

  `BadParameter`

  If an external identifier and the respective internal identifier were given at the same time, e.g. `Chain` and `ChainLabel`. It is no problem of course to mix for instance `Residue` and `AtomNumber`.

- `->Write`

  `NoFile`

  If no file name has been given to `->Write`

  `IOError`

  If the file could not be opened, e.g. due to a permission problem.

15.2 Methods for Error Handling

- `->Warning`

  Returns true, if a warning has been issued
  Returns false, if no warning has been issued

  ```
  if ($PDB->Warning) { print "Uh-oh...\n" }
  ```

- `->GetWarnings`

  Only returns the warning messages as array (no automatic output)

  ```
  if ($PDB->Warning) {
  @Warnings = $PDB->GetWarnings;
     print @Warnings;
  }
  ```

- `->PrintWarnings`

  Prints out all warning messages and also returns an array.

  ```
  if ($PDB->Warning) {
      @Warning = $PDB->PrintWarnings;
  }
  ```

To check for specific warnings, e.g. to tell the user user explicitely, why his crappy file is crap indeed, one can use the following methods.

e.g. `if ($PDB->Warning_NoChainLabel) { print "Watch out!" }`

- `->Warning_NoENDMDL`

  If a MODEL without a corresponding ENDMDL has been found

- `->Warning_NoChainLabel`

  If no chain ID is given at all

- `->Warning_MultipleChainLabel`

  If a certain chain ID has been found more than once. This can lead to problems when using letters to read chains (`->Get (Chain => "B")`). If this warning has been reported, `->WriteChains` ignores the setting of `->ChainLabelAsLetter` and uses numbers.

- ->Warning_UnknownModel

  If the requested model is not defined

- ->Warning_UnknownChain

  If the requested chain is not defined

- ->Warning_UnknownChainLabel

  If the given chain ID could not be found in the PDB

- ->Warning_UnknownAminoAcid

  If a 1- or 3-letter-code given to AminoAcidConvert has not been recognized

- ->Warning_InvalidAminoAcid

  If a code given to AminoAcidConvert has not 1 or 3 letters

The parser provides the possibility to handle errors via the try/catch/otherwise methodology. The possible errors are divided into IO (error opening, closing, writing a file, etc.), Config (wrong parameters given to a routine) and PDB (error due to crappy PDB format). The syntax for the try/catch block is as follows. It is quite picky, note the semicolon at the end and that no semicolon is after the other blocks (it is actually only one single commmand).

```
try {
   $PDB = ParsePDB->new (FileName => $File);
   $PDB->Parse;
}
catch Exception::Config with {
   $Error = shift;
   print "Error parsing the file:\n $Error";
}
catch Exception::IO with {
   $Error = shift;
   print "An I/O Error occurred:\n $Error";
}
catch Exception::PDB with {
   $Error = shift;
   print "Error parsing the file:\n $Error";
}
finally {
  exit (1);
};
```